理解
Linux进程

Linux. Go. Docker

Table of Contents

iii. 再次感谢

理解Linux进程

关于这本书

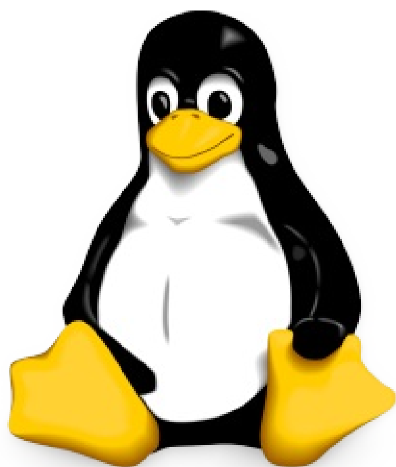
本书受理解*Unix*进程启发而作，用极简的篇幅深入学习进程知识。

理解*Linux*进程用Go重写了所有示例程序，通过循序渐进的方法介绍Linux进程的工作原理和一切你所需要知道的概念。

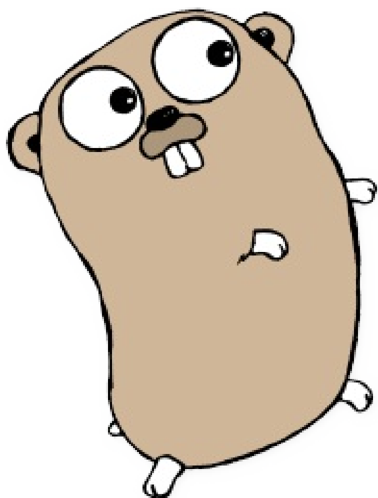
本书适合所有Linux程序员阅读。在线阅读，PDF下载。

三位好朋友

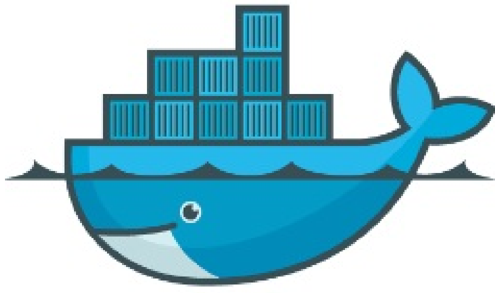
阅读前介绍三位即将与大家打交道的小伙伴：Linux、Go和Docker。



Linux是我们主要的研究对象，书中所有概念与程序都基于Linux，这同样适用于所有Unix-like系统。



Go是本书所有示例程序的实现语言，当然进程的概念与原理是相通的，你也可以使用其他编程语言实现。



Docker为我们创造可重复的实验环境，使用Docker容器你可以轻易地模拟与本书一模一样的运行环境。

Thanks *Wawa Leung*

Otherwise the book would be released two years ago

本书概述

进程的概念大家都很熟悉，但你是否能准确说出僵尸进程的含义呢？还有COW(Copy On Write)、Flock(File Lock)、Epoll和Namespace的概念又是否了解过呢？

本书汇集了进程方方面面的基础知识，加上编程实例，保证阅读后能自如地回答以上问题，在项目开发中对进程的优化也有更深的理解。

本书架构

本书按循序渐进的方式介绍进程的基础概念和拓展知识，主要涵盖以下几个方面。

- 进程的基础知识介绍
- 进程相关的编程实例
- 进程的进阶知识详解
- 项目Run的进程管理
- 使用进程的注意事项

其中项目Run是Go实现的脚本管理工具，通过研究Run的源码能够加深对进程管理的理解。

关于勘误

本书所有内容都托管到GitHub，如果纰漏或错误请提Issue。

示例程序

本书所有示例程序都基于Go编写，代码托管到GitHub。



每章的示例都是可直接运行的Go源文件，例如第一章的Hellow World程序可以通过 `go run hello_world.go` 来运行并查看运行结果。

```
# go run hello_world.go
Hello World
```

接下来介绍使用Docker来运行本书的示例程序。

Docker简介



Docker是一个容器运行平台，你可以将程序及其依赖打包成容器，在不同机器上运行可得到一致的运行效果。因为不同的系统环境或Go版本可能影响程序的运行结果，为了得到可预测、可重复的实验环境，我们引入了Docker容器技术。

Docker使用

我们不仅开源了示例代码，还创建了官方Docker镜像。

只要执行命令 `docker run -i -t tobegit3hub/understand_linux_process_examp` ，就可以马上创建本书的实验环境。进入容器后可以轻易地运行示例程序。

```
root@6a8e36a53495:/go/src# go run hello_world.go
Hello World
```

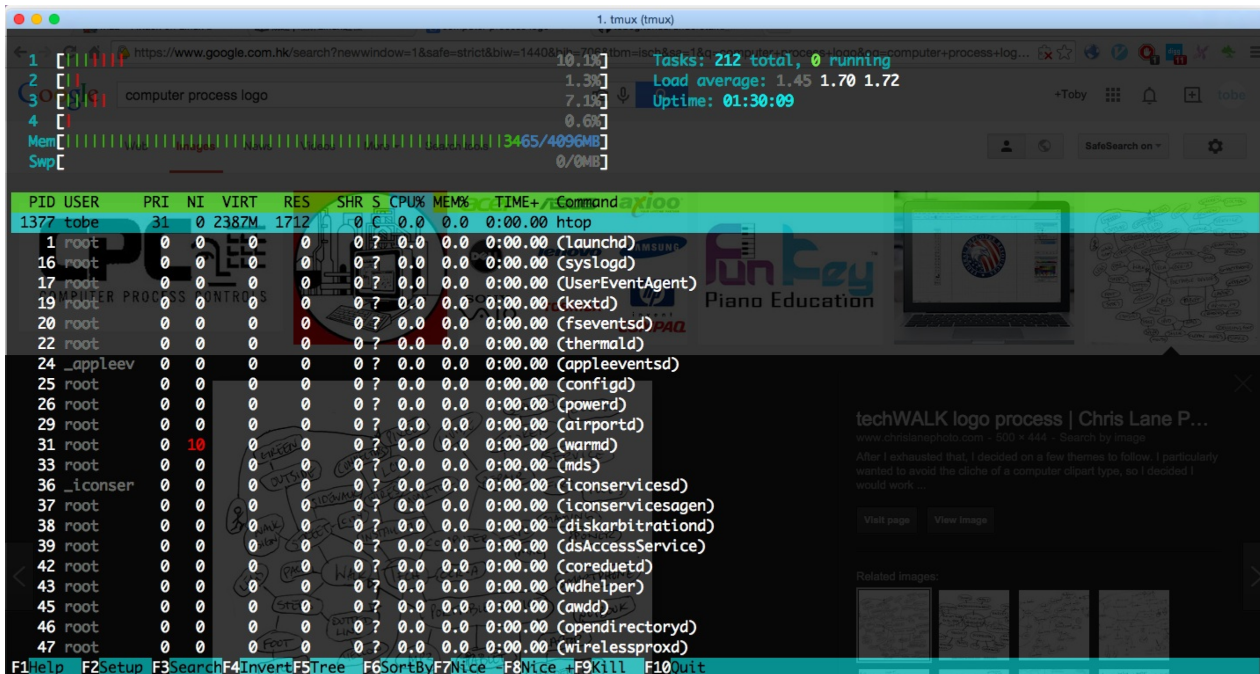
当然你也可以在本地运行自己的Go示例，或者使用官方Go镜像 `docker run -i -t golang:1.4 /bin/bash` 。

第一章 进程基础

作为本书的第一部分，主要介绍进程的PID、进程状态、退出码和POSIX等基础概念。

网络有很多零散的资料介绍基础了，为什么还要花篇幅介绍这些呢？首先我们要保证看过这些章节的都能掌握这些概念，其次通过编写代码实例，我们还能动手验证这些概念，已经不能更赞了。

学习完这章我们应该能够准确回答出PID、PPID、进程名字、进程参数、进程状态、退出码、死锁、活锁、POSIX、Nohup等概念。



进程的定义

根据维基百科的定义，进程(Process)是计算机中已运行程序的实体。用户下达运行程序的命令后，就会产生进程。进程需要一些资源才能完成工作，如CPU使用时间、存储器、文件以及I/O设备，且为依序逐一进行，也就是每个CPU核心任何时间内仅能运行一项进程。

我们简单总结下，进程就是代码运行的实体。这里补充一点，进程不一定都是正在运行的，也可能在等待调度或者停止，进程状态将在后续详细介绍。

举个例子

进程的概念应该很好理解，因为我们都在写代码，这些代码跑起来了就是一个进程，为了完整性我们介绍最简单的的Hello World进程。

Hello World进程

Hello World程序是每门编程语言的入门示例，注意这个程序还不是进程哦，它的作用是在终端输出“Hello World”然后直接退出。

当我们运行Hello World程序时，系统就创建一个Hello World进程。这也是最简单的进程了，没有系统调用、进程间通信等，输出字符串后就退出了。

Bash实现

用Bash实现Hello World程序只需要一行代码，运行后新的进程也可以输出“Hello World”，然后就没有然后了。

```
root@87096bf68cb2:/go/src# echo Hello World
Hello World
```

稍微提一下 `echo` 是Linux自带的程序，可以接受一个或多个参数，反正就是如实地把它们输出到终端而已。

这样最简单的Linux进程就诞生了，当然我们也可以用Go重写Hello World程序。

Go实现

Go实现的程序源码可参见hello_world.go。

```
package main

import (
  "fmt"
)

func main() {
  fmt.Println("Hello World")
}
```

运行后得到以下的输出。

```
root@87096bf68cb2:/go/src# go run hello_world.go
Hello World
```

Hello World进程运行时究竟发生了什么，接下来我们将从各个方面介绍进程的概念。

PID

首先我们来学习PID这个概念，PID全称Process ID，是标识和区分进程的ID，它是一个全局唯一的正整数。

原来Hello World进程运行时也有一个PID，只是它运行结束后PID也释放了，我们可以通过print_pid.go程序显示当前进程的PID。

示例程序

程序print_pid.go的源码如下，通过 `Getpid()` 函数可以获得当前进程的PID。

```
package main

import (
  "fmt"
  "os"
)

func main() {
    fmt.Println(os.Getpid())
}
```

运行结果

```
root@87096bf68cb2:/go/src# go run print_pid.go
2922
root@87096bf68cb2:/go/src# go run print_pid.go
2932
```

可以看出，进程运行时PID是由操作系统随机分配的，同一个程序运行两次会产生两个进程，当然也就有两个不同的PID。

那PID究竟有什么用呢？我们稍后会讨论，现在先了解下PPID。

PPID

每个进程除了一定有PID还会有PPID，也就是父进程ID，通过PPID可以找到父进程的信息。

为什么进程都会有父进程ID呢？因为进程都是由父进程衍生出来的，后面会详细介绍几种衍生的方法。那么跟人类起源问题一样，父进程的父进程的父进程又是什么呢？实际上有一个PID为1的进程是由内核创建的init进程，其他子进程都是由它衍生出来，所以前面的描述并不准确，进程号为1的进程并没有PPID。

因为所有进程都来自于一个进程，所以Linux的进程模型也叫做进程树。

示例程序

要想获得进程的PPID，可以通过以下 `Getppid()` 这个函数来获得，print_ppid.go程序的代码如下。

```
package main

import (
  "fmt"
  "os"
  )

func main() {
    fmt.Println(os.Getppid())
}
```

运行结果

```
root@87096bf68cb2:/go/src# go run print_ppid.go
2892
root@87096bf68cb2:/go/src# go run print_ppid.go
2902
```

有趣的事情发生了，有没有发现每次运行的父进程ID都不一样，这不符合我们的预期啊，原来我们通过 `go run` 每次都会启动一个新的Go虚拟机来执行进程。

编译后运行

如果我们先生成二进制文件再执行结果会怎样呢？

```
root@87096bf68cb2:/go/src# ./print_ppid
1
root@87096bf68cb2:/go/src# ./print_ppid
1
root@87096bf68cb2:/go/src# ps aux |grep "1" |grep -v "ps" |grep -v "grep"
root         1  0.0  0.3  20228  3184 ?        Ss   07:25   0:00 /bin/bash
```

这次我们发现父进程ID都是一样的了，而且通过 `ps` 命令可以看到父进程就是 `bash` ，说明通过终端执行命令其实是从 `bash` 这个进程衍生出各种子进程。

为了执行这个程序要查找包依赖、编译、打包、链接(和 `go build` 做一样的东西)然后执行，这是全新的进程。

拿到PID和PPID后有什么用呢？马上揭晓。

查看PID

首先我们想知道进程的PID，可以通过`top`或者`ps`命令来查看。

Top

在命令行执行`top`后，得到类似下面的输出，可以看到目前有三个进程，PID分别是1、8和9。

```
top - 12:45:18 up 1 min,  0 users,  load average: 0.86, 0.51, 0.20
Tasks:   3 total,   1 running,   2 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   2056748 total,   301984 used,  1754764 free,    20984 buffers
KiB Swap:  1427664 total,        0 used,  1427664 free.   231376 cached Mem

PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
1 root      20   0    4312    692    612 S   0.0  0.0   0:00.23 sh
8 root      20   0   20232   3048   2756 S   0.0  0.1   0:00.03 bash
9 root      20   0   21904   2384   2060 R   0.0  0.1   0:00.00 top
```

PS

执行`ps aux`后输出如下，其中`aux`参数让`ps`命令显示更详细的参数信息。前面PID为9的top进程已经退出了，取而代之的是PID为11的ps进程。

```
root@fa13d0439d7a:/go/src# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.2  0.0   4312   692 ?        Ss   12:45   0:00 /bin/sh -c /bin/bash
root         8  0.0  0.1  20232  3224 ?        S    12:45   0:00 /bin/bash
root        11  0.0  0.0  17484  2000 ?        R+   12:46   0:00 ps aux
```

使用PID

拿到PID后，我们就可以通过`kill`命令来结束进程了，也可以通过`kill -9`或其他数字向进程发送不同的信号。

信号是个很重要的概念，我们后面会详细介绍，那么有了进程ID，我们也可以看看进程名字。

进程名字

每个进程都一定有进程名字，例如我们运行`top`，进程名就是“top”，如果是自定义的程序呢？

其实进程名一般都是进程参数的第一个字符串，在Go中可以这样获得进程名。

```
package main

import (
  "fmt"
  "os"
)

func main() {
    processName := os.Args[0]

    fmt.Println(processName)
}
```

进程的输出结果如下。

```
root@87096bf68cb2:/go/src# go run process_name.go
/tmp/go-build650749614/command-line-arguments/_obj/exe/process_name
root@87096bf68cb2:/go/src# go build process_name.go
root@87096bf68cb2:/go/src# ./process_name
./process_name
```

是否稍稍有些意外，因为`go run`会启动进程重新编译、链接和运行程序，因此每次运行的进程名都不相同，而编译出来的程序有明确的名字，所以它的进程的名字都是一样的。

知道这些以后，我们可以开始接触接进程的运行参数。

进程参数

任何进程启动时都可以赋予一个字符串数组作为参数，一般名为ARGV或ARGS。

通过解析这些参数可以让你的程序更加通用，例如 `cp` 命令通过给定两个参数就可以复制任意的文件，当然如果需要的参数太多最好还是使用配置文件。

获得进程Argument

进程参数一般可分为两类，一是Argument，也就是作为进程运行的实体参数。例如 `cp config.yml config.yml.bak` 的这两个参数。

设计Go程序时可以轻易地获得这些参数，argument.go代码如下，代码来自https://gobyexample.com/command-line-arguments。

```
package main

import "os"
import "fmt"

func main() {
  argsWithProg := os.Args
  argsWithoutProg := os.Args[1:]

  arg := os.Args[3]
  fmt.Println(argsWithProg)
  fmt.Println(argsWithoutProg)
  fmt.Println(arg)
}
```

运行效果如下。

```
$ go build command-line-arguments.go
$ ./command-line-arguments a b c d
[./command-line-arguments a b c d]
[a b c d]
c
```

可以看出通过 `os.Args` ，不管是不是实体参数都可以获得，但是对于类似开关的辅助参数，Go提供了另一种更好的方法。

获得进程Flag

使用Flag可以更容易得将命令行参数转化成我们需要的数据类型，其中flag.go代码如下，代码来自https://gobyexample.com/command-line-flags。

```
package main

import "flag"
import "fmt"

func main() {
  wordPtr := flag.String("word", "foo", "a string")
  numbPtr := flag.Int("numb", 42, "an int")
  boolPtr := flag.Bool("fork", false, "a bool")
  var svar string
  flag.StringVar(&svar, "svar", "bar", "a string var")

  flag.Parse()

  fmt.Println("word:", *wordPtr)
  fmt.Println("numb:", *numbPtr)
```

```
  fmt.Println("fork:", *boolPtr)
  fmt.Println("svar:", svar)
  fmt.Println("tail:", flag.Args())
}
```

运行结果如下，相比直接使用 `os.Args` 代码也简洁了不少。

```
root@87096bf68cb2:/go/src# ./flag -word=opt -numb=7 -fork -svar=flag
word: opt
numb: 7
fork: true
svar: flag
tail: []
root@87096bf68cb2:/go/src# ./flag -h
Usage of ./flag:
  -fork=false: a bool
  -numb=42: an int
  -svar="bar": a string var
  -word="foo": a string
```

使用Go获取进程参数是很简单的，不过一旦参数太多，最佳实践还是使用配置文件。

进程参数只有在启动进程时才能赋值，如果需要在程序运行时进行交互，就需要了解进程的输入与输出了。

进程输入与输出

每个进程操作系统都会分配三个文件资源，分别是标准输入(STDIN)、标准输出(STDOUT)和错误输出(STDERR)。通过这些输入流，我们能够轻易得从键盘获得数据，然后在显示器输出数据。

标准输入

来自管道(Pipe)的数据也是标准输入的一种，我们写了以下的实例来输出标注输入的数据。

```
package main

import (
  "fmt"
  "io/ioutil"
  "os"
)

func main() {
  bytes, err := ioutil.ReadAll(os.Stdin)
  if err != nil {
    panic(err)
  }

  fmt.Println(string(bytes))
}
```

运行结果如下。

```
root@87096bf68cb2:/go/src# echo string_from_stdin | go run stdin.go
string_from_stdin
```

标准输出

通过 `fmt.Println()` 把数据输出到屏幕上，这就是标准输出了，这里不太演示了。

错误输出

程序的错误输出与标准输出类似，这里暂不演示。

了解完进程一些基础概念，我们马上要深入学习并发与并行的知识了。

并发与并行

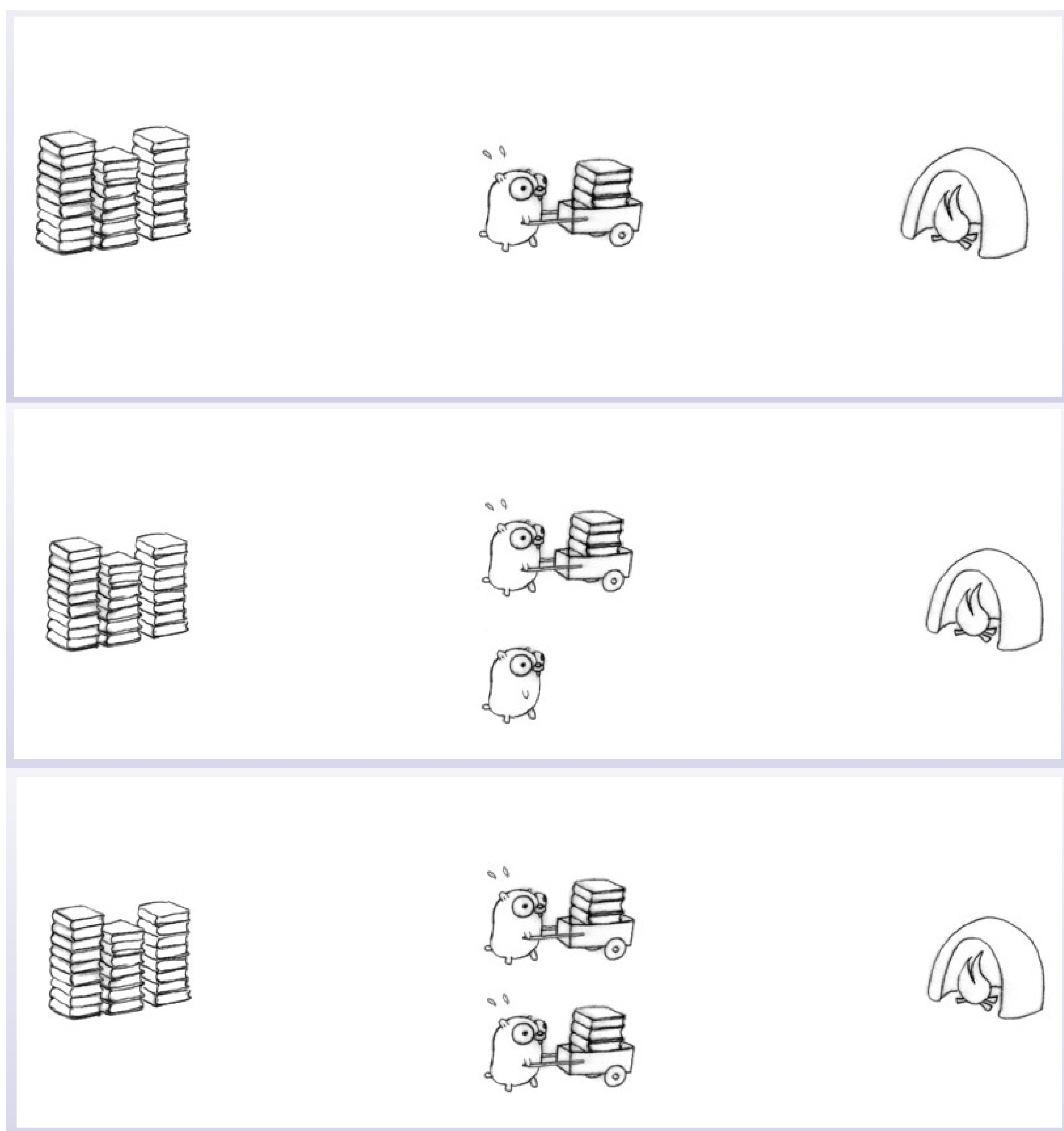
并发(Concurrently)和并行(Parallel)是两个不同的概念。借用Go创始人Rob Pike的说法，并发不是并行，并发更好。并发是一共要处理(deal with)很多事情，并行是一次可以做(do)多少事情。

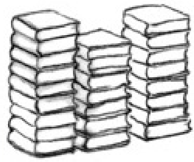
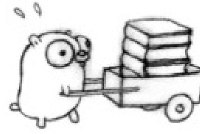
举个简单的例子，华罗庚泡茶，必须有烧水、洗杯子、拿茶叶等步骤。现在我们想尽快做完这件事，也就是"一共要处理很多事情"，有很多方法可以实现并发，例如请多个人同时做，这就是并行。并行是实现并发的一种方式，但不是唯一的方式。我们一个人也可以实现并发，例如先烧水、然后不用等水烧开就去洗杯子，所以通过调整程序运行方式也可以实现并发。

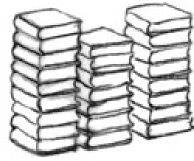
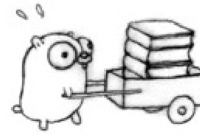
大神讲解

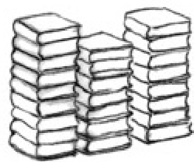
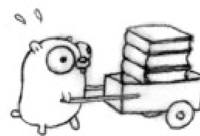
如果还不理解，建议看Rob Pike题为Concurrency is not Parallelism的演讲PPT和演讲视频。

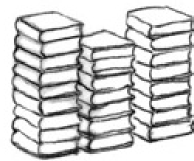
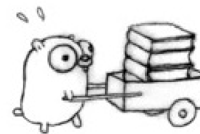
我把演讲的PPT截图贴出来方便大家理解。

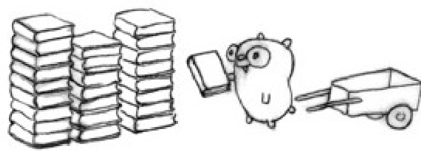
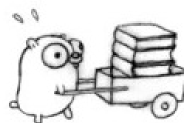
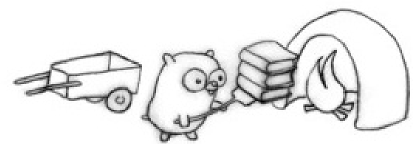


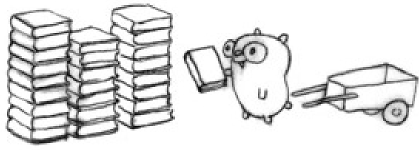
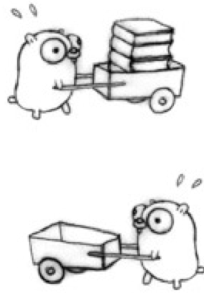
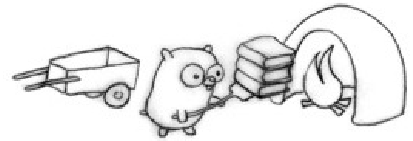
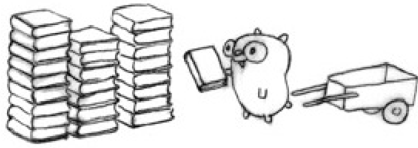
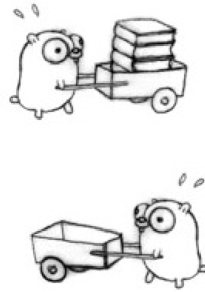
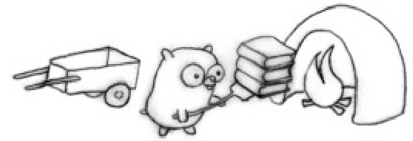
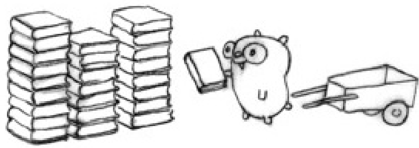
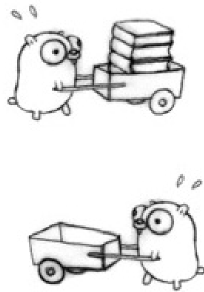
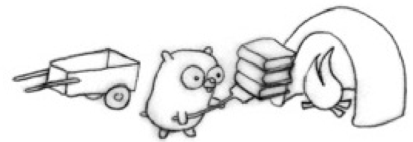
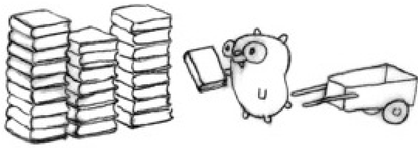
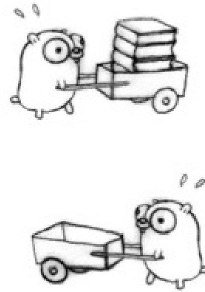
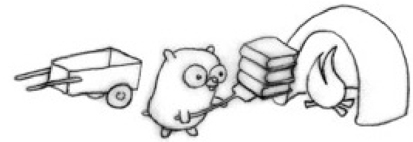
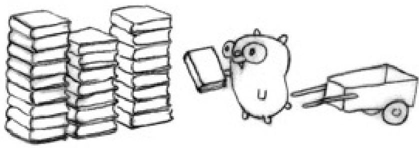
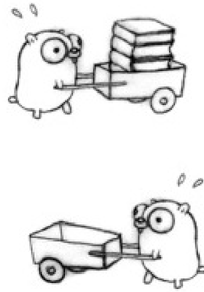
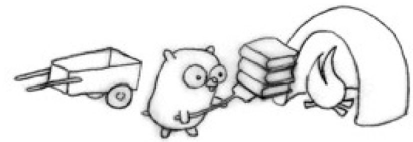



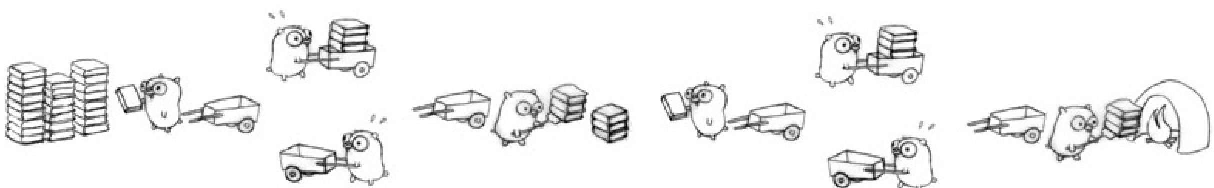



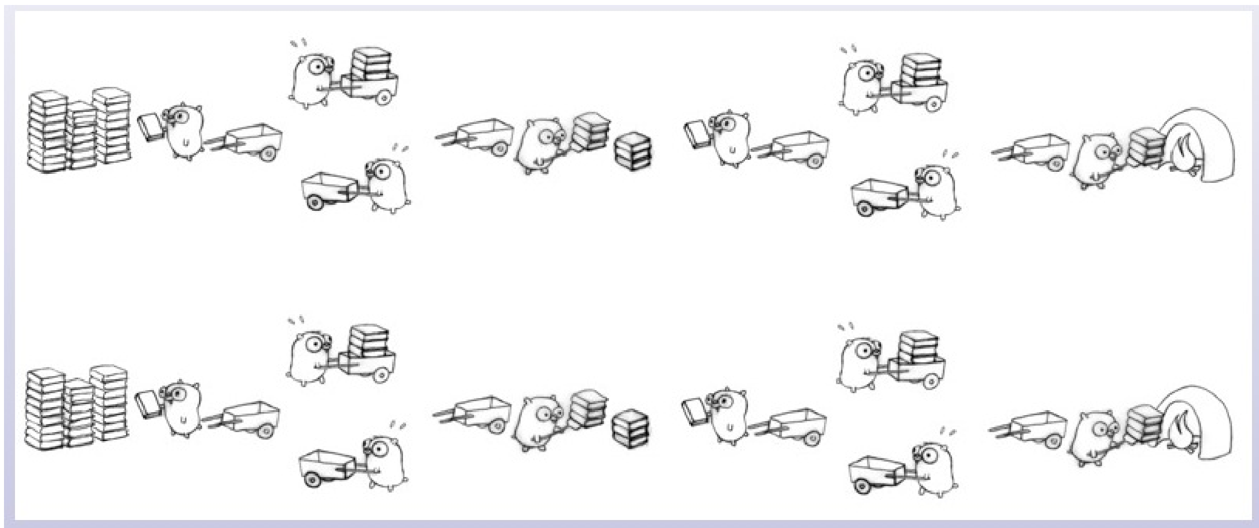







总结

总结一下，并行是实现并发的一种方式，在多核CPU的时代，并行是我们设计高效程序所要考虑的，那么进程是不是越多越好呢？

进程越多越好？

前面提到多进程的并行可以提高并发度，那么进程是越多越好？一般遇到这种问题都回答不是，事实上，很多大型项目都不会同时开太多进程。

下面以支持100K并发量的Nginx服务器为例。

举个例子: Nginx

Nginx是一个高性能、高并发的Web服务器，也就是说它可以同时处理超过10万个HTTP请求，而它建议的启动的进程数不要超过CPU个数，为什么呢？

我们首先要知道Nginx是Master-worker模型，Master进程只负责管理Worker进程，而Worker进程是负责处理真实的请求。每个Worker进程能够处理的请求数跟内存有关，因为在Linux上Nginx使用了epoll这种多路复用的IO接口，所以不需要多线程做并行也能实现并发。

而多进程有一个坏处就是带来了CPU上下文切换时间，所以一味提高进程个数反而使系统系能下降。当然如果当前进程小于CPU个数，就没有充分利用多核的资源，所以Nginx建议Worker数应该等于CPU个数。

特殊情况

我们想想进程数应该等于CPU数，但是如果进程有阻塞呢？这是是应该提高进程数增加并行数的。

在Nginx的例子中，如果Nginx主要负责静态内容的下载，而服务器内存比较小，大部分文件访问都需要读磁盘，这时候进程很容易阻塞，所以建议提高下Worker数目。

绑定CPU

一般情况下除了确保进程数等于CPU数，我们还可以绑定进程与CPU，这就保证了最少的CPU上下文切换。

在Nginx中可以这样配置。

```
worker_processes 4;
worker_cpu_affinity 1000 0100 0010 0001;
```

这是通过系统调用sched_setaffinity()实现了，感兴趣大家可以自行学习这方面的知识。

通过这个例子大家对进程的并发与并行应该有更深入的理解，接下来了解下进程状态的概念。

进程状态

根据进程的定义，我们知道进程是代码运行的实体，而进程有可能是正在运行的，也可能是已经停止的，这就是进程的状态。

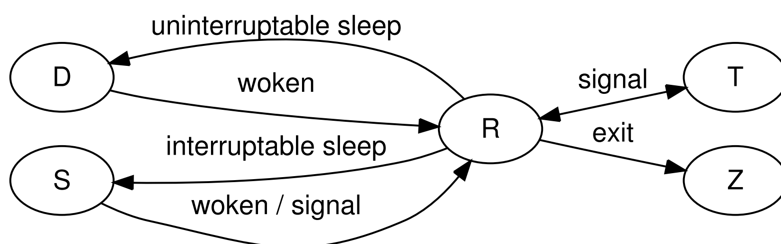
网上有人总结进程一共5种状态，也有总结是8种，究竟应该怎么算呢，最好的方法还是看Linux源码。进程状态的定义在fs/proc/array.c文件中。

```
/*
* The task state array is a strange "bitmap" of
* reasons to sleep. Thus "running" is zero, and
* you can test for combinations of others with
* simple bit tests.
*/
static const char * const task_state_array[] = {
  "R (running)",        /*   0 */
  "S (sleeping)",        /*   1 */
  "D (disk sleep)",    /*   2 */
  "T (stopped)",        /*   4 */
  "t (tracing stop)",    /*   8 */
  "X (dead)",        /*  16 */
  "Z (zombie)",        /*  32 */
};
```

这真的是Linux的源码，可以看出进程一共7种状态，含义也比较清晰，注意其中D(disk sleep)称为不可中断睡眠状态(uninterruptible sleep)。

知道进程状态本身没什么

进程状态转换



使用Ptrace

include/linux/sched.h

```
struct task_struct {
  volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
  void *stack;
  atomic_t usage;
  unsigned int flags;    /* per process flags, defined below */
  unsigned int ptrace;
```

查看状态

通过 `ps aux` 可以看到进程的状态。

O：进程正在处理器运行,这个状态从来没有见过.
S：休眠状态（sleeping）
R：等待运行（runable）R Running or runnable (on run queue) 进程处于运行或就绪状态

I：空闲状态（idle）
Z：僵尸状态（zombie）
T：跟踪状态（Traced）
B：进程正在等待更多的内存页
D: 不可中断的深度睡眠，一般由IO引起，同步IO在做读或写操作时，cpu不能做其它事情，只能等待，这时进程处于这种状态，如果程序采用异步IO，这种状态应该就很少见到了

其中就绪状态表示进程已经分配到除CPU以外的资源，等CPU调度它时就可以马上执行了。运行状态就是正在运行了，获得包括CPU在内的所有资源。等待状态表示因等待某个事件而没有被执行，这时候不耗CPU时间，而这个时间有可能是等待IO、申请不到足够的缓冲区或者在等待信号。

状态转换

进程的运行过程也就是进程状态转换的过程。

例如就绪状态的进程只要等到CPU调度它时就马上转为运行状态，一旦它需要的IO操作还没有返回时，进程状态也就转换成等待状态。

进程状态间转换还有很多，这里不一一细叙，马上去学习进程退出码吧。

退出码

任何进程退出时，都会留下退出码，操作系统根据退出码可以知道进程是否正常运行。

退出码是0到255的整数，通常0表示正常退出，其他数字表示不同的错误。

示例程序

```
package main

func main() {
  panic("Call panic()")
}
```

运行结果

```
root@fa13d0439d7a:/go/src# go run exit_code.go
panic: Call panic()

goroutine 16 [running]:
runtime.panic(0x425900, 0xc208000010)
/usr/src/go/src/pkg/runtime/panic.c:279 +0xf5
main.main()
/go/src/exit_code.go:4 +0x61

goroutine 17 [runnable]:
runtime.MHeap_Scavenger()
/usr/src/go/src/pkg/runtime/mheap.c:507
runtime.goexit()
/usr/src/go/src/pkg/runtime/proc.c:1445

goroutine 18 [runnable]:
bgsweep()
/usr/src/go/src/pkg/runtime/mgc0.c:1976
runtime.goexit()
/usr/src/go/src/pkg/runtime/proc.c:1445
exit status 2
```

我们可以看到最后一行输出了`exit status 2`，证明进程的退出码是2，也就是异常退出。相比之下，运行Hello World程序并没有输出退出码，也就是进程正常结束了。

使用退出码

不管是正常退出还是异常退出，进程都结束了这个退出码有意义吗？

当然有意义，我们在写Bash脚本时，可以根据前一个命令的退出码选择是否执行下一个命令。例如安装Run程序的命令`wget https://github.com/runscripts/run-release/blob/master/0.3.6/linux_amd64/run && sudo run --init`，只有下载脚本成功才会执行后面的安装命令。

Travis CI是为开源项目提供持续集成的网站，因为测试脚本是由开发者写的，Travis只能通过测试脚本的返回值来判断这次测试是否正常通过。

Docker使用Dockerfile来构建镜像，这是类似Bash的领域定义语言(DSL)，每一行执行一个命令，如果命令的进程退出码不为0，构建镜像的流程就会中止，证明Dockerfile有异常，方便用户排查问题。

了解进程退出码后，我们去看更多的进程资源。

进程文件

在Linux中“一切皆文件”，进程的一切运行信息(占用CPU、内存等)都可以在文件系统找到，例如看一下PID为1的进程信息。

```
root@87096bf68cb2:/go/src# ls /proc/1/
attr        cmdline          cwd      fdinfo   loginuid   mounts      numa_maps      pagemap      sessionid  status    v
auxv        comm             environ  gid_map  maps       mountstats  oom_adj        personality  smaps      syscall
cgroup      coredump_filter  exe      io       mem        net         oom_score      projid_map   stat       task
clear_refs  cpuset           fd       limits   mountinfo  ns          oom_score_adj  root         statm      uid_map
```

我们可以看一下它的运行状态，通过 `cat /proc/1/status` 即可。

```
root@87096bf68cb2:/go/src# cat /proc/1/status
Name:   bash
State:  S (sleeping)
Tgid:   1
Ngid:   0
Pid:    1
PPid:   0
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 256
Groups:
VmPeak:    20300 kB
VmSize:    20300 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:      3228 kB
VmRSS:      3228 kB
VmData:      408 kB
VmStk:       136 kB
VmExe:       968 kB
VmLib:      2292 kB
VmPTE:        60 kB
VmSwap:        0 kB
Threads:        1
SigQ:   0/3947
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: 0000000000010000
SigIgn: 0000000000380004
SigCgt: 000000004b817efb
CapInh: 00000000a80425fb
CapPrm: 00000000a80425fb
CapEff: 00000000a80425fb
CapBnd: 00000000a80425fb
Seccomp:        0
Cpus_allowed:   1
Cpus_allowed_list:      0
Mems_allowed:   00000000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        684
nonvoluntary_ctxt_switches:     597
```

参考Linux手册可以看到更多信息，我们这不再深究，实际上 `ps` 命令获得的数据也是在这个文件系统获得的。

我们已经了解了这么多进程属性，是时候开始学习“传说中”的死锁问题了。

死锁概念

死锁(Deadlock)就是一个进程拿着资源A请求资源B，另一个进程拿着资源B请求资源A，双方都不释放自己的资源，导致两个进程都进行不下去。

示例程序

我们可以写代码模拟进程死锁的例子。

```
package main

func main() {
  ch := make(chan int)
  <-ch
}
```

运行结果

```
root@fa13d0439d7a:/go/src# go run deadlock.go
fatal error: all goroutines are asleep - deadlock!

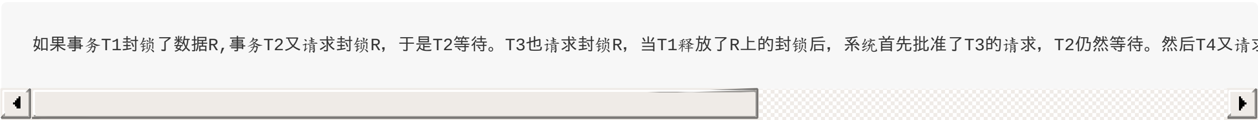
goroutine 16 [chan receive]:
main.main()
/go/src/deadlock.go:5 +0x4f
exit status 2
```

这里Go虚拟机已经替我们检测出死锁的情况，因为所有Goroutine都阻塞住没有运行，关于Goroutine的概念有机会详细介绍一下。

我们可能很早就接触过死锁的概念，也很容易模拟出来，那么你是否知道活锁呢？

活锁概念

相对于死锁，活锁(Livelock)是什么概念呢？有意思的是，百度百科把这个解释错了。



```
如果事务T1封锁了数据R,事务T2又请求封锁R，于是T2等待。T3也请求封锁R，当T1释放了R上的封锁后，系统首先批准了T3的请求，T2仍然等待。然后T4又请
```

这显然是饿死(Starvation)的定义，进入活锁的进程是没有阻塞的，会继续使用CPU，但外界看到整个进程都没有前进。

活锁实例

举个很简单的例子，两个人相向过独木桥，他们同时向一边谦让，这样两个人都过不去，然后二者同时又移到另一边，这样两个人又过不去了。如果不受其他因素干扰，两个人一直同步在移动，但外界看来两个人都没有前进，这就是活锁。

活锁会导致CPU耗尽的，解决办法是引入随机变量、增加重试次数等。

所以活锁也是程序设计上可能存在的问题，导致进程都没办法运行下去了，还耗CPU。

接下来介绍本章最大的内容，POSIX。

POSIX简介

POSIX(Portable Operation System Interface)听起来好高端，就是一种操作系统的接口标准，至于谁遵循这个标准呢？就是大名鼎鼎的Unix和Linux了，有人问Mac OS是否兼容POSIX呢，答案是Yes苹果的操作系统也是Unix-based的。

有了这个规范，你就可以调用通用的API了，Linux提供的POSIX系统调用在Unix上也能执行，因此学习Linux的底层接口最好就是理解POSIX标准。

补充一句，目前很多编程语言(Go、Java、Python、Ruby等)都是天生跨平台的，因此我们很少注意系统调用的兼容性。实际上POSIX提供了这些语言上跨平台的语义，而且这是源码级别的保证。

POSIX规范

POSIX是一些IEEE标准，包括1003.0、1003.1、1003.1b和2003等，实际上连Linux也没有完全兼容这些定义，不过只用Linux来学习POSIX足够了。

鉴于绝大多数程序员都没看过IEEE文档，我们就翻一下IEEE 1003.1-2001吧。

IEEE Std 1003.1™-2001/Cor 1-2002

The Open Group Technical Standard
Base Specifications, Issue 6

1003.1™-2001/Cor 1-2001

Standard for Information Technology —
Portable Operating System Interface (POSIX®)

Technical Corrigendum 1

Sponsor

Portable Applications Standards Committee
of the
IEEE Computer Society

and

The Open Group



THE Open GROUP

篇幅跟论文差不多，大意就是修正Base标准存在的问题，这个文档没有增加新的接口，但是加了符号、非函数的定义和保留更多命名空间。这是非常严谨的文档，感兴趣的同学可以读下，对普通的程序员我们还是以下的内容。

POSIX进程

我们运行Hello World程序时，操作系统通过POSIX定义的`fork`和`exec`接口创建起一个POSIX进程，这个进程就可以使用通用的IPC、信号等机制。

POSIX线程

POSIX也定义了线程的标准，包括创建和控制线程的API，在Pthreads库中实现，有关线程的知识有机会再深入学习。

Nohup命令

每个开发者都会躺过这个坑，在命令行跑一个后台程序，关闭终端后发现进程也退出了，网上搜一下发现要用 `nohup`，究竟什么原因呢？

原来普通进程运行时默认会绑定TTY(虚拟终端)，关闭终端后系统会给上面所有进程发送TERM信号，这时普通进程也就退出了。当然还有些进程不会退出，这就是后面将会提到的守护进程。

`Nohup` 的原理也很简单，终端关闭后会给此终端下的每一个进程发送SIGHUP信号，而使用 `nohup` 运行的进程则会忽略这个信号，因此终端关闭后进程也不会退出。

举个例子

我们用Go实现最简单的Web服务器，代码web_server.go如下。

```
package main

import (
  "fmt"
  "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Handle request")
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8000", nil)
}
```

然后在终端上运行，并测试一下。

```
➜ go build web_server.go
➜ ./web_server &
[1] 25967
➜  wget 127.0.0.1:8000
--2014-12-28 22:24:07--  http://127.0.0.1:8003/
Connecting to 127.0.0.1:8003... connected.
HTTP request sent, awaiting response... 200 OK
Length: 5 [text/plain]
Saving to: 'index.html.4'

100%[=====================>] 5           --.-K/s   in 0s

2014-12-28 22:24:07 (543 KB/s) - 'index.html' saved [5/5]
```

如果关闭终端，`curl` 命令就连不上我们的Web服务器了。如果使用 `nohup` 运行呢？

```
➜ go build web_server.go
➜ nohup ./web_server &
[1] 25968
➜ exit
➜ wget 127.0.0.1:8003
--2014-12-28 22:24:11--  http://127.0.0.1:8003/
```

发现关闭终端对Web服务器进程没有任何影响，这正是我们预期的。这是运行守护进程最简单的方法，实际上标准的守护进程除了处理信号外，还要考虑这种因素，后面将会详述。

创建进程

本章开始时演示了Hello World程序，其实已经创建了新的进程，通过`Bash`或者`zsh`这些Shell很容易创建新的进程，但Shell本身是怎么实现的呢？我们又能不能用Go实现类似Shell的功能呢？

系统调用

原来这一切都是操作系统给我们做好的，然后暴露了使用的API接口，这就是系统调用。Linux或者其他Unix-like系统都提供了`fork()`和`exec()`等接口，Bash或者我们写的程序都可以通过调用这些接口来操作进程。

Go创建进程

而Go已经封装了与进程相关的接口，主要在`os/exec`这个Package中。通过使用封装好的接口，我们很容易就可以在自己的项目中调用其他进程了。

这一章已经介绍了这么多概念，马上会有实践环节，用Go实现多种方式来来创建和运行外部进程。

第二章 Go编程实例

学习完进程基础知识，我们通过几个Go编程实例介绍如果使用Go运行外部进程。

这章主要是编程练习，学习完这章后对进程的使用和Go对进程的使用应该都有更深的理解。

衍生(Spawn)新进程

这是来自GoByExample的例子，代码在https://gobyexample.com/spawning-processes。

它能够执行任意Go或者非Go程序，并且等待放回结果，外部进程结束后继续执行本程序。

代码实现

```
package main

import "fmt"
import "io/ioutil"
import "os/exec"

func main() {
    dateCmd := exec.Command("date")
    dateOut, err := dateCmd.Output()
    if err != nil {
        panic(err)
    }
    fmt.Println("> date")
    fmt.Println(string(dateOut))

    grepCmd := exec.Command("grep", "hello")
    grepIn, _ := grepCmd.StdinPipe()
    grepOut, _ := grepCmd.StdoutPipe()
    grepCmd.Start()
    grepIn.Write([]byte("hello grep\ngoodbye grep"))
    grepIn.Close()
    grepBytes, _ := ioutil.ReadAll(grepOut)
    grepCmd.Wait()
    fmt.Println("> grep hello")
    fmt.Println(string(grepBytes))

    lsCmd := exec.Command("bash", "-c", "ls -a -l -h")
    lsOut, err := lsCmd.Output()
    if err != nil {
        panic(err)
    }
    fmt.Println("> ls -a -l -h")
    fmt.Println(string(lsOut))
}
```

运行结果

```
$ go run spawning-processes.go
> date
Wed Oct 10 09:53:11 PDT 2012
> grep hello
hello grep
> ls -a -l -h
drwxr-xr-x  4 mark 136B Oct 3 16:29 .
drwxr-xr-x 91 mark 3.0K Oct 3 12:50 ..
-rw-r--r--  1 mark 1.3K Oct 3 16:28 spawning-processes.go
```

归纳总结

因此如果你的程序需要执行外部命令，可以直接使用 `exec.Command()` 来Spawn进程，并且根据需要获得外部程序的返回值。

执行(Exec)外部程序

这是来自GoByExample的例子，代码在https://gobyexample.com/execing-processes。

把新程序加载到自己的内存。

与Spawn不同，执行外部程序并不会返回到原进程中，也就是让外部程序完全取代本进程。

代码实现

```
package main

import "syscall"
import "os"
import "os/exec"

func main() {
    binary, lookErr := exec.LookPath("ls")
    if lookErr != nil {
        panic(lookErr)
    }
    args := []string{"ls", "-a", "-l", "-h"}
    env := os.Environ()
    execErr := syscall.Exec(binary, args, env)
    if execErr != nil {
        panic(execErr)
    }
}
```

运行结果

```
$ go run execing-processes.go
total 16
drwxr-xr-x  4 mark 136B Oct 3 16:29 .
drwxr-xr-x 91 mark 3.0K Oct 3 12:50 ..
-rw-r--r--  1 mark 1.3K Oct 3 16:28 execing-processes.go
```

归纳总结

如果你的程序就是用来执行外部程序的，例如后面提到的项目实例Run，那使用 `syscall.Exec` 执行外部程序就最合适了。注意调用该函数后，本进程后面的代码将不可能再执行了。

复制(Fork)进程

如果我们仅仅想复制父进程的堆栈空间呢，很遗憾Go没有提供这样的接口，因为使用Spawn、Exec和Goroutine已经能覆盖绝大部分的使用案例了。

事实上无论是Spawn还是Exec都是通过实现Fork系统调用来实现的，后面将会详细介绍它的实现原理。

第三章 进程进阶

学习进程基础和Go编程时候后，我们会接触进程更底层的概念，包括信号、进程锁和系统调用等。

通过学习这章我们对进程的所有概念都了如指掌了，充分理解这些概念后有助于我们实现更高效的应用程序。

```
1. tmux (tmux)
Processes: 212 total, 2 running, 11 stuck, 199 sleeping, 962 threads                                   22:53:10
Load Avg: 1.84, 1.79, 1.75  CPU usage: 2.66% user, 2.91% sys, 94.41% idle  SharedLibs: 12M resident, 16M data, 0B linkedit.
MemRegions: 31469 total, 2060M resident, 106M private, 471M shared. PhysMem: 4034M used (598M wired), 59M unused.
VM: 671G vsize, 1066M framework vsize, 0(0) swapins, 0(0) swapouts. Networks: packets: 215797/296M in, 186055/62M out.
Disks: 184818/5643M read, 60202/1694M written.

PID   COMMAND      %CPU TIME     #TH  #WQ  #PORT MEM    PURG   CMPRS  PGRP PPID STATE    BOOSTS     %CPU_ME %CPU_OTHRS UID  FAULTS  COW
1368  screencaptur 0.4  00:00.30 2    0    46-   2088K+ 20K    0B     302  302  sleeping *0[12]     0.00000 0.43450    501  11792+  361+
1367  QuickLookSat 0.0  00:00.33 2    0    40    3224K  0B     0B     1367 1    sleeping  0[0]      0.00000 0.00000    501  2429    178
1366  quicklookd   0.0  00:00.18 8    4    108   3728K  0B     0B     1366 1    stuck     0[3]      0.00000 0.00000    501  3142    236
1364  top          2.6  00:01.15 1/1  0    22    2332K  0B     0B     1364 805  running  *0[1]      0.00000 0.00000    0    60170+  98
1363  ocspd        0.0  00:00.02 1    0    16    916K   0B     0B     1363 1    sleeping *0[1]      0.00000 0.00000    0    901     106
1352  ssh-agent    0.0  00:00.03 3    0    44    1212K  0B     0B     1352 1    sleeping *0[1]      0.00000 0.00000    501  1076    132
1175  Google Chrom 0.0  00:01.03 13   0    83    18M    0B     0B     273  273  sleeping *0[6]      0.00000 0.00000    501  12783   1602
1170  Google Chrom 0.1  00:11.35 13   0    128   98M    3440K  0B     273  273  sleeping *0[27]     0.00000 0.00000    501  65811   1495
1136  Google Chrom 1.0  00:47.87 13   0    128   160M   104M   0B     273  273  sleeping *0[28]     0.00000 0.00000    501  206323  1526
1118  Atom Helper  0.0  00:02.29 6    0    60    28M    0B     0B     276  391  sleeping *0[6]      0.00000 0.00000    501  17394   1026
941   spindump     0.0  00:02.33 2    1    43    13M    0B     108M   941  1    sleeping *0[1]      0.00000 0.00000    0    141637  168
903-  SogouInput   0.0  00:19.98 3    0    178   21M    12K    11M    903  1    sleeping *0[4205]   0.00000 0.00000    501  93363   1890
902   imklaunchage 0.0  00:00.04 2    0    64    1012K  0B     872K   902  1    sleeping *0[1]      0.00000 0.00000    501  1603    133
898   mdworker     0.0  00:08.07 3    0    53    1740K  0B     328K   898  1    sleeping *0[1]      0.00000 0.00000    501  15211   195
897   node         2.0  02:09.44 8    0    68    109M   0B     9908K  897  722  sleeping *0[1]      0.00000 0.00000    501  72976   251
869   systemstatsd 0.0  00:00.22 2    1    22    1468K  0B     52K    869  1    sleeping  0[27]     0.00000 0.00000    0    3038    71
867   mdflagwriter 0.0  00:00.14 2    1    20    1196K  0B     8192B  867  1    sleeping *0[1]      0.00000 0.00000    501  1251    77
805   zsh          0.0  00:00.81 1    0    15    6600K  0B     0B     805  658  sleeping *0[1]      0.00000 0.00000    501  22881   5012
796   discoveryd   0.0  00:00.94 19   17   105   3472K  0B     32K    796  1    sleeping *0[1]      0.00000 0.00000    65   2731    145
787   com.apple.Ch 0.0  00:00.03 2    0    36    632K   0B     1552K  787  1    sleeping  0[3]      0.00000 0.00000    501  2090    133
785   com.apple.cm 0.0  00:00.01 2    0    30    248K   0B     776K   785  1    sleeping  0[1]      0.00000 0.00000    0    1633    132
784   ScopedBookma 0.0  00:00.04 2    1    32    656K   0B     1028K  784  1    sleeping  0[1]      0.00000 0.00000    501  1260    132
775   distnoted    0.0  00:00.02 2    0    29    564K   0B     124K   775  1    sleeping *0[1]      0.00000 0.00000    89   606     90
765   mdworker     0.0  00:04.66 4    0    55    6384K  0B     812K   765  1    sleeping *0[1]      0.00000 0.00000    501  34664   243
764   mdworker     0.0  00:05.04 4    0    55    6244K  0B     924K   764  1    sleeping *0[1]      0.00000 0.00000    501  37842   247
```

进程锁

这里的进程锁与线程锁、互斥量、读写锁和自旋锁不同，它是通过记录一个PID文件，避免两个进程同时运行的文件锁。

进程锁的作用之一就是可以协调进程的运行，例如crontab使用进程锁解决冲突提到，使用crontab限定每一分钟执行一个任务，但这个进程运行时间可能超过一分钟，如果不用进程锁解决冲突的话两个进程一起执行就会有问题。后面提到的项目实例Run也有类似的问题，通过进程锁可以解决进程间同步的问题。

使用PID文件锁还有一个好处，方便进程向自己发停止或者重启信号。Nginx编译时可指定参数 `--pid-path=/var/run/nginx.pid` ，进程起来后就会把当前的PID写入这个文件，当然如果这个文件已经存在了，也就是前一个进程还没有退出，那么Nginx就不会重新启动。进程管理工具Supervisord也是通过记录进程的PID来停止或者拉起它监控的进程的。

使用进程锁

进程锁在特定场景是非常适用的，而操作系统默认不会为每个程序创建进程锁，那我们该如何使用呢？

其实要实现一个进程锁很简单，通过文件就可以实现了。例如程序开始运行时去检查一个PID文件，如果文件存在就直接退出，如果文件不存在就创建一个，并把当前进程的PID写入文件中。这样我们很容易可以实和读锁，但是所有流程都需要自己控制。

当然根据DRY(Don't Repeat Yourself)原则，Linux已经为我们提供了 `flock` 接口。

使用Flock

Flock提供的是advisory lock，也就是建议性的锁，其他进程实际上也可以读写这个锁文件。Linux上可以直接使用 `flock` 命令，使用C可以调用原生的 `flock` 接口，这里详细介绍Go 1.3引入的 `FcntlFock()` 。

我们封装了简单的接口。

```
// Control the lock of file.
func fcntlFlock(lockType int16, path ...string) error {
  var err error
  if lockType != syscall.F_UNLCK {
    mode := syscall.O_CREAT | syscall.O_WRONLY
    lockFile, err = os.OpenFile(path[0], mode, 0666)
    if err != nil {
      return err
    }
  }

  lock := syscall.Flock_t{
    Start:  0,
    Len:    1,
    Type:   lockType,
    Whence: int16(os.SEEK_SET),
  }
  return syscall.FcntlFlock(lockFile.Fd(), syscall.F_SETLK, &lock)
}
```

这样对进程加锁。

```
// Lock the file.
func Flock(path string) error {
  return fcntlFlock(syscall.F_WRLCK, path)
}
```

这样对进程解锁。

```
// Unlock the file.
func Funlock(path string) error {
  err := fcntlFlock(syscall.F_UNLCK)
  if err != nil {
    return err
  } else {
    return lockFile.Close()
  }
}
```

学习完进程锁，我们开始了解各种进程，如孤儿进程、僵尸进程。

孤儿进程概念

我们经常听别人说到孤儿进程(Orphan Process)，究竟是什么呢，现在我们一次理解透。

根据维基百科的解释，孤儿进程指的是在其父进程执行完成或被终止后仍继续运行的一类进程。

孤儿进程与僵尸进程是完全不同的，后面会详细介绍僵尸进程。而孤儿进程借用了现实中孤儿的概念，也就是父进程不在了，子进程还在运行，这时我们就把子进程的PPID设为1。前面讲PID提到，操作系统会创建进程号为1的init进程，它没有父进程也不会退出，可以收养系统的孤儿进程。

作用

在现实中用户可能刻意使进程成为孤儿进程，这样就可以让它与父进程会话脱钩，成为后面会介绍的守护进程。

僵尸进程

当一个进程完成它的工作终止之后，它的父进程需要调用wait()或者waitpid()系统调用取得子进程的终止状态。

一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程。

理解了孤儿进程和僵尸进程，我们临时加了守护进程这一小节，守护进程就是后台进程吗？没那么简单。

守护(Daemon)进程

我们可以认为守护进程就是后台服务进程，因为它会有一个很长的生命周期提供服务，关闭终端不会影响服务，也就是说可以忽略某些信号。

实现守护进程

首先要保证进程在后台运行，可以在启动程序后面加 `&` ，当然更原始的方法是进程自己 `fork` 然后结束父进程。

```
if (pid=fork()) {
  exit(0); // Parent process
}
```

然后是与终端、进程组、会话(Session)分离。每个进程创建时都绑定一个终端，而且属于一个进程组(进程组也有GID不过等同进程组长的PID)，这些进程组在一个会话中，如果是子进程一般会从父进程继承这些信息，想要与环境分离可以使用以下的系统调用。

```
setsid();
```

同样地我们会从父进程继承文件掩码(mask)，可以手动清理掩码。

```
umask(0);
```

如果需要我们可以改变当前工作目录，避免运行时必须使用当前所在的文件系统。

使用Nohup

前面提到过 `nohup` 命令，是让程序以守护进程运行的方式之一，程序运行后忽略SIGHUP信号，也就说关闭终端不会影响进程的运行。

类似的命令还有 `disown` ，这里不再详述。

进程间通信

IPC全称Interprocess Communication，指进程间协作的各种方法，当然包括共享内存，信号量或Socket等。

管道(Pipe)

管道是进程间通信最简单的方式，任何进程的标准输出都可以作为其他进程的输入。

信号(Signal)

下面马上会介绍。

消息队列(Message)

和传统消息队列类似，但是在内核实现的。

共享内存(Shared Memory)

后面也会有更详细的介绍。

信号量(Semaphore)

信号量本质上是一个整型计数器，调用`wait`时计数减一，减到零开始阻塞进程，从而达到进程、线程间协作的作用。

套接口(Socket)

也就是通过网络来通信，这也是最通用的IPC，不要求进程在同一台服务器上。

信号

我们知道信号是进程间通信的其中一种方法，当然也可以是内核给进程发送的消息，注意信息只是告诉进程发生了什么事件，而不会传递任何数据。

这是进程这个概念设计时就考虑到的了，因为我们希望控制进程，就像一个小孩我们想他按我们的想法做，前提就是他能够接受信号并且理解信号的含义。

信号种类

Linux中定义了很多信号，不同的Unix-like系统也不一样，我们可以通过下面的命令来查当前系统支持的种类。

```
➜ kill -l
HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV SYS PIPE ALRM TERM URG STOP TSTP CONT CHLD TTIN TTOU IO XCPU XFSZ VTAL
```

其中1至31的信号为传统UNIX支持的信号，是不可靠信号(非实时的)，32到63的信号是后来扩充的，称做可靠信号(实时信号)。不可靠信号和可靠信号的区别在于前者不支持排队，可能会造成信号丢失，而后者不会。

简单介绍几个我们最常用的，在命令行中止一个程序我们一般摁Ctrl+c，这就是发送SIGINT信号，而使用kill命令呢？默认是SIGTERM，加上 `-9` 参数才是SIGKILL。

编程实例

```
import os/signal

siganl.Notify()
signal.Stop()
```

这是Go封装的信号接口，我们可以以此实现一个简单的信号发送和处理程序。

系统调用

我们要想启动一个进程，需要操作系统的调用(system call)。实际上操作系统和普通进程是运行在不同空间上的，操作系统进程运行在内核态(todo: kernel space)，开发者运行得进程运行在用户态(todo: user space)，这样有效规避了用户程序破坏系统的可能。

如果用户态进程想执行内核态的操作，只能通过系统调用了。Linux提供了超多系统调用函数，我们关注与进程相关的系统调用后面也会详细讲解。

文件描述符

Linux很重要的设计思想就是一切皆文件，网络是文件，键盘等外设也是文件，很神奇吧？于是所有资源都有了统一的接口，开发者可以像写文件那样通过网络传输数据，我们也可以通过 `/proc/` 的文件看到进程的资源使用情况。

内核给每个访问的文件分配了文件描述符(File Descriptor)，它本质是一个非负整数，在打开或新建文件时返回，以后读写文件都要通过这个文件描述符了。

应用

我们想想操作系统打开的文件这么多，不可能他们共用一套文件描述符整数吧？这样想就对了，Linux实现时这个fd其实是一个索引值，指向每个进程打开文件的记录表。

POSIX已经定义了STDIN_FILENO、STDOUT_FILENO和STDERR_FILENO三个常量，也就是0、1、2。这三个文件描述符是每个进程都有的，这也解释了为什么每个进程都有编号为0、1、2的文件而不会与其他进程冲突。

文件描述符帮助应用找到这个文件，而文件的打开模式等上下文信息存储在文件对象中，这个对象直接与文件描述符关联。

限制

注意了，每个系统对文件描述符个数都有限制。我们网上看到配置 `ulimit` 也是为了调大系统的打开文件个数，因为一般服务器都要同时处理成千上万个起请求，记住socket连接也是文件哦，使用系统默认值会出现莫名奇怪的问题。

讲文件描述符其实是为高深莫测的epoll做铺垫，掌握epoll对进程已经有很深的理解了。

简介

Epoll是poll的改进版，更加高效，能同时处理大量文件描述符，跟高并发有关，Nginx就是充分利用了epoll的特性。讲这些没用，我们先了解poll是什么。

Poll

Poll本质上是Linux系统调用，其接口为`int poll(struct pollfd *fds,nfds_t nfds, int timeout)`，作用是监控资源是否可用。

举个例子，一个Web服务器建了多个socket连接，它需要知道里面哪些连接传输发了请求需要处理，功能与`select`系统调用类似，不过`poll`不会清空文件描述符集合，因此检测大量socket时更加高效。

Epoll

我们重点看看epoll，它大幅提升了高并发服务器的资源使用率，相比poll而言哦。前面提到poll会轮询整个文件描述符集合，而epoll可以做到只查询被内核IO事件唤醒的集合，当然它还提供边沿触发(Edge Triggered)等特性。

不知大家是否了解C10K问题，指的是服务器如何支持同时一万个连接的问题。如果是一万个连接就有至少一万个文件描述符，poll的效率也随文件描述符的更加而下降，epoll不存在这个问题是因为它仅关注活跃的socket。

实现

这是怎么做到的呢？简单来说epoll是基于文件描述符的callback函数来实现的，只有发生IO时间的socket会调用callback函数，然后加入epoll的Ready队列。更多实现细节可以参考Linux源码，

Mmap

无论是select、poll还是epoll，他们都要把文件描述符的消息送到用户空间，这就存在内核空间和用户空间的内存拷贝。其中epoll使用mmap来共享内存，提高效率。

Mmap不是进程的概念，这里提一下是因为epoll使用了它，这是一种共享内存的方法，而Go语言的设计宗旨是"不要通过共享来通信，通过通信来共享"，所以我们也可以思考下进程的设计，是使用mmap还是Go提供的channel机制呢。

共享内存

对于共享内存是好是坏，我们不能妄下定论，不过学习一下总是好的。

不同进程之间内存空间是独立的，也就是说进程不能访问也不会干扰其他进程的内存。如果两个进程希望通过共享内存的方式通信呢？可以通过`mmap()`系统调用实现。

Go实例

Go也实现了`mmap()`函数支持共享内存，不过也是通过cgo来调用C实现的系统调用函数。Cgo是什么？它是Go调用C语言模块的功能，当然这种调用很可能是平台相关的，也就是无法保证在Windows也能正确运行。

具体代码参见Golang对共享内存的操作，有时间我们也愿意写一个更简单易懂的例子。

写时复制(Copy On Write)

一般我们运行程序都是Fork一个进程后马上执行Exec加载程序，而Fork的是否实际上用的是父进程的堆栈空间，Linux通过Copy On Write技术极大地减少了Fork的开销。

Copy On Write的含义是只有真正写的时候才把数据写到子进程的数据，Fork时只会把页表复制到子进程，这样父子进程都指向同一个物理内存页，只有再写子进程的时候才会把内存页的内容重新复制一份。

Cgroups

Cgroups全称Control Groups，是Linux内核用于资源隔离的技术。目前Cgroups可以控制CPU、内存、磁盘访问。

使用

Cgroups是在Linux 2.6.24合并到内核的，不过项目在不断完善，3.8内核加入了对内存的控制(kmemcg)。

要使用Cgroups非常简单，阅读前建议看sysadmincasts的视频，https://sysadmincasts.com/episodes/14-introduction-to-linux-control-groups-cgroups。

我们首先在文件系统创建Cgroups组，然后修改这个组的属性，启动进程时指定加入的Cgroups组，这样进程相当于在一个受限的资源内运行了。

实现

Cgroups的实现也不是特别复杂。有一个特殊的数据结构记录进程组的信息。

有人可能已经知道Cgroups是Docker容器技术的基础，另一项技术也是大名鼎鼎的Namespaces。

Namespaces简介

Linux Namespaces是资源隔离技术，在2.6.23合并到内核，而在3.12内核加入对用户空间的支持。

Namespaces是容器技术的基础，因为有了命名空间的隔离，才能限制容器之间的进程通信，像虚拟内存对于物理内存那样，开发者无需针对容器修改已有的代码。

使用Namespaces

阅读以下教程前建议看看，https://blog.jtlebi.fr/2013/12/22/introduction-to-linux-namespaces-part-1-uts/。

Linux内核提供了`clone`系统调用，创建进程时使用`clone`取代`fork`即刻创建同一命名空间下的进程。

更多参数建议`man clone`来学习。

第四章 项目实例Run

Run是开源的脚本管理工具，官方网站http://runscripts.org，项目地址https://github.com/runscripts/run。

Run可以执行任意的脚本，当然使用到Go库提供的系统调用程序。

Run项目架构

Run是一个命令行工具，没有复杂的CS或BS架构，只是通过解析命令行或者配置文件来下载运行相应的脚本。

Flock

Run使用了前面提到的进程文件锁，避免同时运行同一个脚本。同时运行同一个脚本会有什么问题呢？例如我们`run pt-summary`，同时另一个终端执行`run -u pt-summary`，这样前一个命令有可以使用旧脚本也可能使用新脚本，这需我们规避这样的问题。

实现Run

实现Flock

前面提到进程的文件锁，实际上Run也用到了，可以试想下以下的场景。

用户A执行 `run pt-summary` ，由于本地已经缓存了所以会直接运行本地的脚本。同时用户B执行 `run -u pt-summary` ，加上 `-u` 或者 `--update` 参数后Run会从远端下载并运行最新的脚本。如果不加文件锁的话，用户A的行为就不可预测了，而文件锁很好得解决了这个问题。

具体使用方法如下，我们封装了以下的接口。

```
var lockFile *os.File

// Lock the file.
func Flock(path string) error {
    return fcntlFlock(syscall.F_WRLCK, path)
}

// Unlock the file.
func Funlock(path string) error {
    err := fcntlFlock(syscall.F_UNLCK)
    if err != nil {
      return err
    } else {
        return lockFile.Close()
    }
}

// Control the lock of file.
func fcntlFlock(lockType int16, path ...string) error {
    var err error
    if lockType != syscall.F_UNLCK {
      mode := syscall.O_CREAT | syscall.O_WRONLY
      mask := syscall.Umask(0)
      lockFile, err = os.OpenFile(path[0], mode, 0666)
      syscall.Umask(mask)
      if err != nil {
        return err
      }
    }

    lock := syscall.Flock_t{
      Start:  0,
      Len:    1,
      Type:   lockType,
      Whence: int16(os.SEEK_SET),
    }
    return syscall.FcntlFlock(lockFile.Fd(), syscall.F_SETLK, &lock)
}
```

在运行脚本前就调用锁进程的方法。

```
// Lock the script.
lockPath := cacheDir + ".lock"
err = flock.Flock(lockPath)
if err != nil {
  utils.LogError("%s: %v\n", lockPath, err)
  os.Exit(1)
}
```

实现HTTP请求

使用Run时它会自动从网上下载脚本，走的HTTP协议，具体实现方法如下。

```go
// Retrieve a file via HTTP GET.
func Fetch(url string, path string) error {
  response, err := http.Get(url)
  if err != nil {
    return err
  }
  if response.StatusCode != 200 {
    return Errorf("%s: %s", response.Status, url)
  }

  defer response.Body.Close()
  body, err := ioutil.ReadAll(response.Body)
  if err != nil {
    return err
  }
  if strings.HasPrefix(url, MASTER_URL) {
    // When fetching run.conf, etc.
    return ioutil.WriteFile(path, body, 0644)
    } else {
      // When fetching scripts.
      return ioutil.WriteFile(path, body, 0777)
    }
}
`
```

Run的总体代码是很简单的，主要是通过解析run.conf下载相应的脚本并执行。

第五章 注意事项

对进程有了深入理解后，我们编写实际应用可能遇到这些坑，这里总结一下。

创建目录权限

如果你想创建一个目录并授予777权限，你需要怎么做？查看Go的API文档我们可以这样写。

源文件为mkdir.go。

```
package main

import (
  "fmt"
  "os"
)

func main() {
    err := os.MkdirAll("/tmp/gotest/", 0777)
    if err != nil {
      panic(err)
    }

    fmt.Println("Mkdir /tmp/gotest/")
}
```

运行结果

```
➜  understand_linux_process_examples git:(master) ✗ ll /tmp/
drwxr-xr-x   2 tobe  wheel    68B Dec 30 10:06 gotest
➜  understand_linux_process_examples git:(master) ✗ umask
022
```

正确做法

代码在mkdir_umask.go中。

```
package main

import (
  "fmt"
  "os"
  "syscall"
)

func main() {
    mask := syscall.Umask(0)
    defer syscall.Umask(mask)

    err := os.MkdirAll("/tmp/gotest/", 0777)
    if err != nil {
      panic(err)
    }

    fmt.Println("Mkdir /tmp/gotest/")
}
```

注意事项

这并不是Go的Bug，包括Linux系统调用都是这样的，创建目录除了给定的权限还要加上系统的Umask，Go也是如实遵循这种约定。

如果你想达到你的预期权限，知道Umask及其用法是必须的。

捕获SIGKILL

SIGKILL是常见的Linux信号，我们使用 `kill` 命令杀掉进程也就是像进程发送SIGKILL信号。

和其他信号不同，SIGKILL和SIGSTOP是不可被Catch的，因此下面的代码是能编译通过但也是无效的，更多细节可以参考golang/go#9463.

```
c := make(chan os.Signal, 1)
signal.Notify(c, syscall.SIGKILL, syscall.SIGSTOP)
```

注意事项

这是Linux内核的限制，这种限制也是为了让操作系统有可能控制进程的生命周期，理解后我们也不应该去尝试捕获SIGKILL。

不过还是有人这样去做，最后结果也不符合预期，这需要我们对底层有足够的理解。

系统调用sendfile

Sendfile是Linux实现的系统调用，可以通过避免文件在内核态和用户态的拷贝来优化文件传输的效率。

其中大名鼎鼎的分布式消息队列服务Kafka就使用sendfile来优化效率，具体用法可参见其官方文档。

优化策略

在普通进程中，要从磁盘拷贝数据到网络，其实是需要通过系统调用，进程也会反复在用户态和内核态切换，频繁的数据传输在此有效率问题。因此我们必须意识到Linux给我们提供了sendfile这样的系统调用，可以提高进程的数据传输效率。

后记

最后一章列举本文参考的过的书籍和项目，欢迎大家补充和讨论更多有关进程的知识。

参考书籍

- 理解Unix进程
- Unix编程艺术
- Unix环境高级编程
- Go Web编程
- Go并发编程实战

项目学习

- Linux
- Go
- Docker
- Run
- GoByExample

Thanks for reading!

理解 Linux进程

Linux. Go. Docker